Northeastern University

DS 5220 – Lecture 2
Linear Regression

Roi Yehoshua

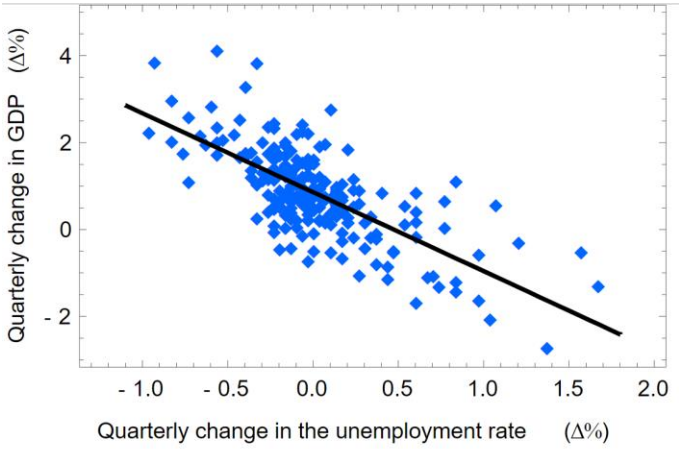
Agenda

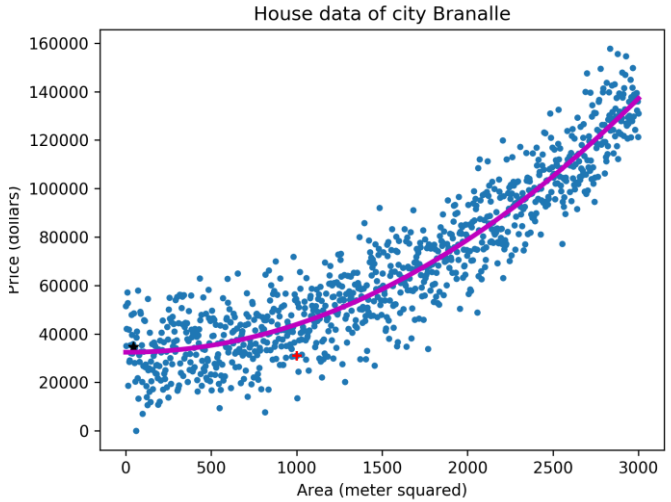
Northeastern University

- Linear regression
- The normal equations
- Gradient descent
- Maximum likelihood estimation
- Probabilistic interpretation of least squares
- Scikit-Learn estimator API

Regression Task Definition

- **Given**: a **training set** of n labeled examples $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$
  - Each $\mathbf{x}_i$ is a d-dimensional vector of feature values, $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})^T$
    - Also known as the **explanatory** variables or the **independent** variables
  - $y_i \in \mathrm{R}$ is continuous-valued output generated by an unknown function $y = f(\mathbf{x})$
    - Also known as the **response** variable or the **dependent** variable
- **Goal**: learn a function h (**hypothesis**) that maps the feature set $\mathbf{x}$ into the target y





Regression Applications Examples

Northeastern University

- Predict a stock market index using other economic indicators
- Project the total sales of a company based on the amount spent on advertising
- Forecast the amount of precipitation in a region based on the characteristics of the jet stream
- Estimate the age of a fossil according to the amount of carbon-14 left in it
- Predict the credit card balance based on applicant's information

Linear Regression

Northeastern University

- In linear regression, we approximate *y* as a linear function of **x**:

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + \ldots + w_2 x_2 + \ldots + w_d x_d$$

- $\mathbf{w} = (w_0, \ldots, w_d)^T$ is a vector of **parameters** (also called **weights**)
- To simplify our notation, we introduce an **intercept term** $x_0 = 1$ so that

$$h(\mathbf{x}) = \sum_{j=0}^{d} w_j x_j = \mathbf{w}^T \mathbf{x}$$

- Our goal is to find the parameters **w** that will make *h*(**x**) as close as possible to *y*
  - at least for the training examples that we have

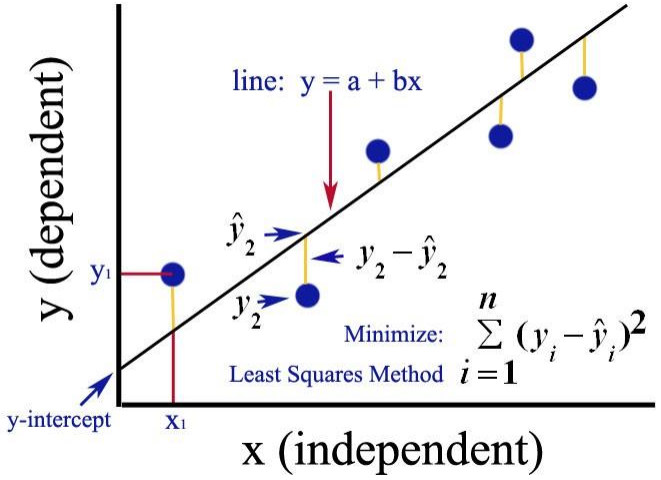
Least Squares Cost Function

Northeastern University

- A **cost function** measures how far a model's predictions are from the true labels
  - Also known as loss or error function
- In linear regression, we define the **least squares** cost function:

$$J(\mathbf{w}) = \sum_{i=1}^{n} (h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2$$

- Minimizing this function gives rise to the **ordinary least squares** method

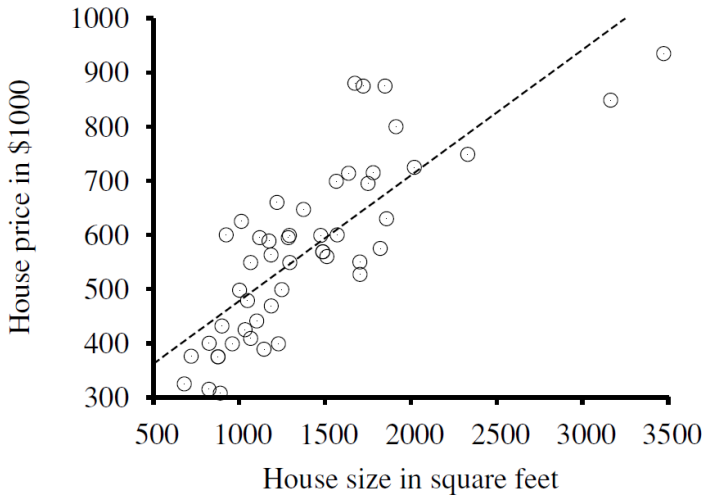


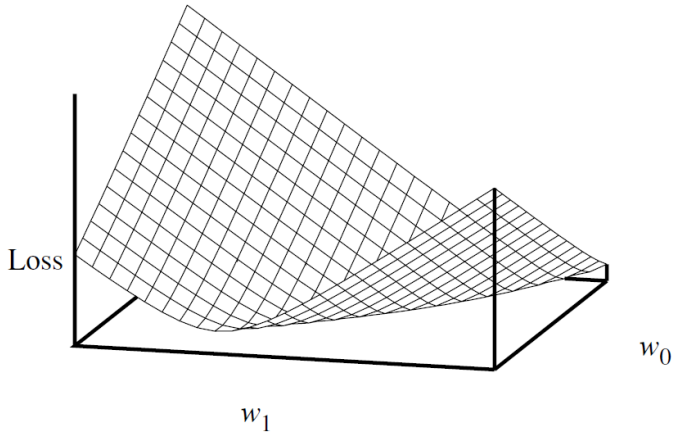
Least Squares Cost Function

- The least squares cost function is **convex**, thus it has a single global minimum
- Reminder: a function $f(x)$ is convex if every chord lies on or above the function

$$f(\lambda a + (1-\lambda)b) \leq \lambda f(a) + (1-\lambda) f(b)$$

  - This is equivalent to the requirement that $f''(x)$ is positive everywhere





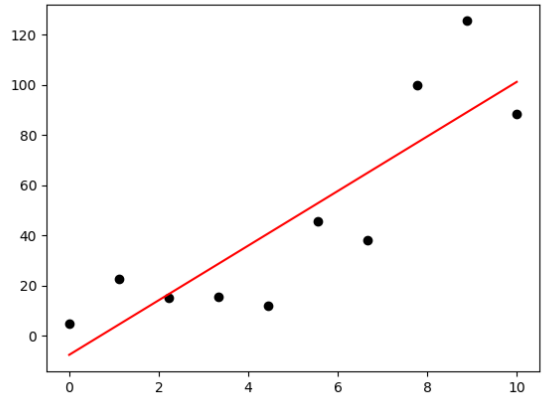
Univariate Linear Regression

Northeastern University

- A linear regression with a single feature *x*
- We are given *n* points: $(x_1, y_1), \ldots, (x_n, y_n)$
- The hypothesis space consists of all the straight lines of the form:

$$h(x) = w_0 + w_1 x$$

  - w_0 is the intercept of the line and w_1 is its slope



- Our goal is to find the line that best fits the training data
- The cost function in this case is:

$$J(\mathbf{w}) = \sum_{i=1}^{n} (h(x_i) - y_i)^2 = \sum_{i=1}^{n} (w_0 + w_1 x_i - y_i)^2$$

- We can find the minimum of $J(\mathbf{w})$ by taking its derivatives with respect to w_0 and w_1 and setting them to 0

Univariate Linear Regression

Northeastern University

- For w_0 we get:

$$\frac{\partial}{\partial w_0}\sum_{i=1}^{n}(w_0 + w_1 x_i - y_i)^2 = \sum_{i=1}^{n}\frac{\partial}{\partial w_0}(w_0 + w_1 x_i - y_i)^2$$

$$= \sum_{i=1}^{n} 2(w_0 + w_1 x_i - y_i)\frac{\partial}{\partial w_0}(w_0 + w_1 x_i - y_i)$$

$$= \sum_{i=1}^{n} 2(w_0 + w_1 x_i - y_i)$$

$$\sum_{i=1}^{n} 2(w_0 + w_1 x_i - y_i) = 0 \Rightarrow n w_0 + w_1 \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i = 0$$

$$\boxed{w_0 = \frac{\sum_{i=1}^{n} y_i - w_1 \sum_{i=1}^{n} x_i}{n}}$$

Univariate Linear Regression

Northeastern University

- For w_1 we get:

$$\frac{\partial}{\partial w_1}\sum_{i=1}^{n}(w_0 + w_1 x_i - y_i)^2 = \sum_{i=1}^{n}\frac{\partial}{\partial w_1}(w_0 + w_1 x_i - y_i)^2$$
$$= \sum_{i=1}^{n} 2(w_0 + w_1 x_i - y_i)\frac{\partial}{\partial w_1}(w_0 + w_1 x_i - y_i)$$
$$= \sum_{i=1}^{n} 2(w_0 + w_1 x_i - y_i)x_i$$

- Setting the derivative equal to 0:

$$\sum_{i=1}^{n} 2(w_0 + w_1 x_i - y_i)x_i = 0$$
$$w_0\sum_{i=1}^{n} x_i + w_1\sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i y_i = 0$$

Univariate Linear Regression

Northeastern University

- Substituting the formula for w_1:

$$\frac{\sum_{i=1}^{n} y_i - w_1 \sum_{i=1}^{n} x_i}{n} \sum_{i=1}^{n} x_i + w_1 \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i y_i = 0$$

$$\frac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n} - w_1 \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n} + w_1 \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i y_i = 0$$

$$w_1 \left[\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}\right] = \sum_{i=1}^{n} x_i y_i - \frac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n}$$

$$w_1 = \frac{n \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

Univariate Linear Regression

Northeastern University

- Therefore, the coefficients of the regression line are:

$$w_1 = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

$$w_0 = \frac{\sum_{i=1}^{n} y_i - w_1 \sum_{i=1}^{n} x_i}{n}$$

Class Exercise

Northeastern University

- Find the regression line that best matches the following data:

x	1	2	3	4
y	2.3	3.9	6.3	7.8

Scikit-Learn Data Sets

Northeastern University

- Scikit-learn comes with a few small standard datasets that can be downloaded using the following functions from sklearn.datasets:

`load_boston`(*[, return_X_y])	Load and return the boston house-prices dataset (regression).
`load_iris`(* [, return_X_y, as_frame])	Load and return the iris dataset (classification).
`load_diabetes`(* [, return_X_y, as_frame])	Load and return the diabetes dataset (regression).
`load_digits`(* [, n_class, return_X_y, as_frame])	Load and return the digits dataset (classification).
`load_linnerud`(* [, return_X_y, as_frame])	Load and return the physical excercise linnerud dataset.
`load_wine`(* [, return_X_y, as_frame])	Load and return the wine dataset (classification).
`load_breast_cancer`(* [, return_X_y, as_frame])	Load and return the breast cancer wisconsin dataset (classification).

  - If return_X_y=True the functions return a tuple with the design matrix *X* and the target *y*
  - Otherwise they return a dictionary with *X*, *y* and also metadata (such as the feature names)

Boston House Prices

Northeastern University

- The Boston house price data set has been used in many ML papers that address regression problems
- The data set contains 506 records, each describing a Boston suburb or town
- The goal is to predict the median value of a house in a given town based on different characteristics of the town, such as crime rate, distance to employment center, etc.
- The data was drawn from the Boston Standard Metropolitan Statistical Area (SMSA) in 1970

Boston House Prices

Northeastern University

- The data set contains the following features:

Number of Instances:	506
Number of Attributes:	13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.
Attribute Information (in order):	• CRIM per capita crime rate by town • ZN proportion of residential land zoned for lots over 25,000 sq.ft. • INDUS proportion of non-retail business acres per town • CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise) • NOX nitric oxides concentration (parts per 10 million) • RM average number of rooms per dwelling • AGE proportion of owner-occupied units built prior to 1940 • DIS weighted distances to five Boston employment centres • RAD index of accessibility to radial highways • TAX full-value property-tax rate per $10,000 • PTRATIO pupil-teacher ratio by town • B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town • LSTAT % lower status of the population • MEDV Median value of owner-occupied homes in $1000's
Missing Attribute Values:	None

Boston House Prices

Northeastern University

- First let's load the data set and examine a few of its records:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.datasets import load_boston
```

```
data = load_boston()

X, y = data.data, data.target
feature_names = data.feature_names
```

```
mat = np.column_stack((X, y))
df = pd.DataFrame(mat, columns=np.append(feature_names, 'MEDV'))
df.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Boston House Prices

Northeastern University

- We can get basic statistical information on the data set using df.describe():

```
df.describe()
```

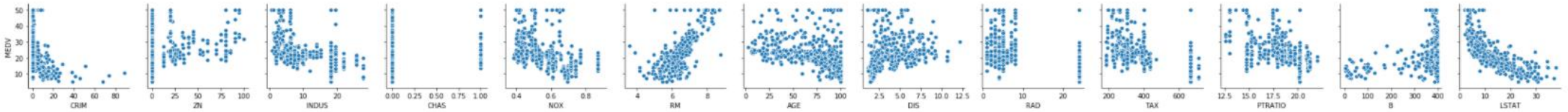
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.0
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.23
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.53
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.0
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.0
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.0
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.0
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.0

Boston House Prices

Northeastern University

- We can plot the pairwise relationships between all the features and the target:

```
sns.pairplot(data=df, y_vars=['MEDV'], x_vars=feature_names);
```



- There seems to be a linear relationship between the average number of rooms (feature 6) and the target variable

Boston House Prices

- Let's try to fit a linear regression model between the average number of rooms and the target variable (the median house price)
- First let's plot the data:

```
def plot_data(x, y):
    plt.scatter(x, y, s=5)
    plt.xlabel('Number of rooms')
    plt.ylabel('Price (in $1000)')
    plt.title('Boston housing prices')
```

```
x = X[:, 5]
plot_data(x, y)
```

Boston House Prices

Northeastern University

- We can find the best regression line by using the equations for w_0 and w_1:

```
n = len(x)
w1 = (n * x.dot(y) - (x.sum() * y.sum())) / (n * (x**2).sum() - x.sum()**2)
w0 = (y.sum() - w1 * x.sum()) / n

w0, w1
```

```
(-34.67062077643899, 9.102108981180377)
```

- The equation of the regression line is:

$$y = -34.671 + 9.102x$$

Boston House Prices

Northeastern University

- To plot the regression line, we can pick any two points along the line
- For example, let's pick the leftmost and rightmost points from the data set:

```
x0 = x.min()
x1 = x.max()
y0 = w0 + w1 * x0
y1 = w0 + w1 * x1
plt.plot([x0, x1], [y0, y1], color='r')
plot_data()
```

Performance Measures

Northeastern University

- Common performance measures for regression models:
  - RMSE (Root Mean Squared Error)
  - MAE (Mean Absolute Error)
  - R^2 score

RMSE

Northeastern University

- The most common performance measure of a regression model is the **Root Mean Square Error** (RMSE)
- RMSE is the square root of the average of the squared errors:

$$\mathrm{RMSE}(\mathbf{w}) = \sqrt{\frac{\sum_{i=1}^{n}(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2}{n}}$$

- RMSE measures the spread of the predicted *y* values around the actual *y* values
- If the errors are normally distributed, you should expect 68% of the predicted values to be within one RMSE, and 95% to be within 2 RMSE of the actual *y* values
  - For example, an RMSE of 5 in the house price means that about 68% of the predictions fall within $5,000 of the actual value, and about 95% fall within $10,000 of the actual value

RMSE

Northeastern University

- You can compute the RMSE directly or by using the function **mean_squared_error()** from sklearn.metrics module and taking the square of it:

```
y_pred = w0 + w1 * x
```

```
rmse = np.sqrt(np.sum((y_pred - y)**2) / n)
print('RMSE:', rmse)
```

```
RMSE: 6.603071389222562
```

```
from sklearn.metrics import mean_squared_error as MSE

rmse = np.sqrt(MSE(y, y_pred))
print('RMSE:', rmse)
```

```
RMSE: 6.603071389222562
```

MAE

Northeastern University

- Mean absolute error (MAE) is the average of absolute errors:

$$\mathrm{MAE}(\mathbf{w}) = \frac{\sum_{i=1}^{n} |h_{\mathbf{w}}(\mathbf{x}_i) - y_i|}{n}$$

- This measure is more robust to outliers than RMSE

```
from sklearn.metrics import mean_absolute_error as MAE

mae = MAE(y, y_pred)
print('MAE:', mae)
```

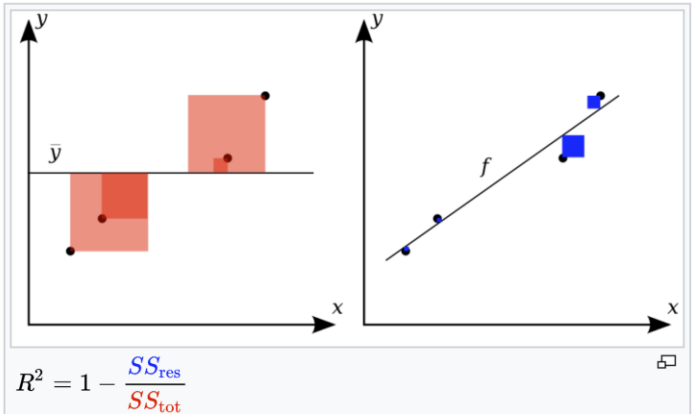
```
MAE: 4.447772901532231
```

R^2 Score

- The **coefficient of determination** R^2 is the proportion of variance in the dependent variable *y* that is predictable from the independent variables in the model

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

  - where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ is the mean of the labels



- Best possible R^2 score is 1.0 (when the model predictions exactly match *y*)
- A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R^2 score of 0.0
- Models that have worse predictions than the baseline will have a negative R^2

R^2 Score

Northeastern University

- You can compute the R^2 score by using the r2_score() function:

```
from sklearn.metrics import r2_score

r2 = r2_score(y, y_pred)
print('R2 score:', r2)
```

```
R2 score: 0.4835254559913341
```

Multiple Linear Regression

Northeastern University

- We now generalize the closed-form solution to the general case
- We begin by rewriting the cost function *J* in matrix-vectorial notation
- We define the **design matrix** *X* to be the $n \times (d + 1)$ matrix that contains the training examples in its rows (including the intercept term)

$$X = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1d} \\ 1 & x_{21} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nd} \end{pmatrix}$$

- Also, let **y** be the *n*-dimensional vector containing all the target values:

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Multiple Linear Regression

Northeastern University

- Since $$h_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{w}^T\mathbf{x}_i = \mathbf{x}_i^T\mathbf{w}$$
- We can easily verify that:

$$X\mathbf{w} - \mathbf{y} = \begin{pmatrix} \mathbf{x}_1^T\mathbf{w} \\ \vdots \\ \mathbf{x}_n^T\mathbf{w} \end{pmatrix} - \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} h_{\mathbf{w}}(\mathbf{x}_1) - y_1 \\ \vdots \\ h_{\mathbf{w}}(\mathbf{x}_n) - y_n \end{pmatrix}$$

- Thus, using the fact that for any vector **z** $\mathbf{z}^T\mathbf{z} = \sum_i z_i^2$, we can write:

$$J(\mathbf{w}) = \sum_{i=1}^{n} (h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2 = (X\mathbf{w} - \mathbf{y})^T(X\mathbf{w} - \mathbf{y})$$

Multivariate Linear Regression

Northeastern University

- To minimize $J(\mathbf{w})$, let's find its derivatives with respect to $\mathbf{w}$:

$$\begin{aligned}\nabla_{\mathbf{w}} J(\mathbf{w}) &= \nabla_{\mathbf{w}} \left((X\mathbf{w}-\mathbf{y})^T(X\mathbf{w}-\mathbf{y})\right) \\ &= \nabla_{\mathbf{w}} \left((X\mathbf{w})^T X\mathbf{w} - (X\mathbf{w})^T\mathbf{y} - \mathbf{y}^T(X\mathbf{w}) + \mathbf{y}^T\mathbf{y}\right) \\ &= \nabla_{\mathbf{w}} \left(\mathbf{w}^T X^T X\mathbf{w} - \mathbf{y}^T(X\mathbf{w}) - \mathbf{y}^T(X\mathbf{w})\right) \\ &= \nabla_{\mathbf{w}} \left(\mathbf{w}^T(X^T X)\mathbf{w} - 2(X^T\mathbf{y})^T\mathbf{w}\right) \\ &= 2X^T X\mathbf{w} - 2X^T\mathbf{y}\end{aligned}$$

We used
- $\nabla_{\mathbf{x}} \mathbf{u}^T\mathbf{x} = \mathbf{u}$
- For symmetric A $\nabla_{\mathbf{x}} \mathbf{x}^T A\mathbf{x} = 2A\mathbf{x}$

- We now set the derivatives to zero, and obtain the **normal equations**:

$$X^T X\mathbf{w} = X^T\mathbf{y}$$

- Thus, the value of $\mathbf{w}$ that minimizes $J(\mathbf{w})$ is given by:

$$\boxed{\mathbf{w}^* = (X^T X)^{-1} X^T\mathbf{y}}$$

Example: Boston House Prices

Northeastern University

- Let's perform linear regression on the Boston housing data set, using all 13 features
- First we split the data set into training and test sets
- To that end, we can use the train_test_split() function from Scikit-Learn:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

```
X_train.shape
```

```
(404, 13)
```

```
X_test.shape
```

```
(102, 13)
```

  - test_size specifies the proportion of the data set to include in the test split (default 0.25)

Example: Boston House Prices

Northeastern University

- We now fit a regression model to the training set using the normal equations:

```
def closed_form_solution(X, y):
    w = np.linalg.inv(X.T @ X) @ X.T @ y
    return w
```

```
X_train_b = np.column_stack((np.ones(len(X_train)), X_train))
w = closed_form_solution(X_train_b, y_train)
w
```

```
array([ 3.80916949e+01, -1.19443447e-01,  4.47799511e-02,  5.48526168e-03,
        2.34080361e+00, -1.61236043e+01,  3.70870901e+00, -3.12108178e-03,
       -1.38639737e+00,  2.44178327e-01, -1.09896366e-02, -1.04592119e+00,
        8.11010693e-03, -4.92792725e-01])
```

Example: Boston House Prices

Northeastern University

- To evaluate the model, we compute RMSE both on the training and test sets:

```
y_train_pred = X_train_b @ w
rmse = np.sqrt(MSE(y_train, y_train_pred))
print('RMSE (train):', rmse)
```

```
RMSE (train): 4.396188144698282
```

```
X_test_b = np.column_stack((np.ones(len(X_test)), X_test))
y_test_pred = X_test_b @ w
rmse = np.sqrt(MSE(y_test, y_test_pred))
print('RMSE (test):', rmse)
```

```
RMSE (test): 5.783509315085026
```

Scikit-Learn Estimator API

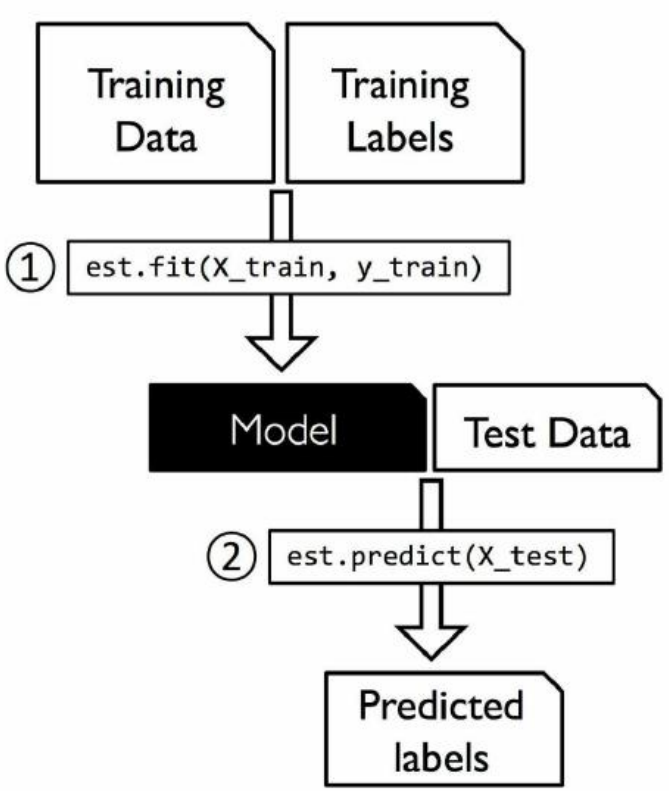
Northeastern University

- The main API implemented by scikit-learn is that of the estimator
- An **estimator** is any object that learns from data
  - It may be a classification, regression or clustering algorithm or a transformer that extracts/filters useful features from raw data
- All estimators expose the method **fit()**, which learns the model's parameters from a given data set
- If the estimator is a **predictor** (capable of making predictions) it also exposes:
  - **predict()** – takes a set of new samples and returns a set of corresponding predictions
  - **score()** – measures the quality of the predictions given a test set
- If the estimator is a **transformer** it also exposes:
  - **transform()** – takes dataset and returns the transformed dataset
  - **fit_transform()** – a convenience method that calls fit() and then transform()

Scikit-Learn Estimator API

Northeastern University

- The typical workflow of using an estimator:



Model Hyperparameters vs. Parameters

Northeastern University

- **Model parameters** are estimated from the data automatically by the ML algorithm
  - e.g., the weights of the regression model
- **Hyperparameters** are meta parameters that are external to the model and whose values are not estimated from the data
  - e.g., the learning rate for training a neural network
- Model hyperparameters are typically set and tuned manually

- In Scikit-learn you specify the hyperparameters in the constructor of the estimator
  - estimator = Estimator(param1=1, param2=2)
- After the data is fitted with an estimator, all the estimated parameters are stored as attributes of the estimator object ending by an underscore
  - estimator.estimated_param_

The LinearRegression Class

Northeastern University

- Scikit-Learn's **LinearRegression** also implements the ordinary least squares method using the normal equations

class `sklearn.linear_model.` **`LinearRegression`**(*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=None*) [source]

Parameter	Description
fit_intercept	Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered).
normalize	This parameter is ignored when fit_intercept is set to False. If True, the data will be normalized before regression by subtracting the mean and dividing by the l2-norm.

- Learned parameters:

Attribute	Description
coef_	Estimated coefficients for the linear regression problem
intercept_	The bias term

The LinearRegression Class

Northeastern University

- The equivalent code using the LinearRegression class looks like this:

```
from sklearn.linear_model import LinearRegression

reg = LinearRegression()
reg.fit(X_train, y_train)
reg.intercept_, reg.coef_
```

```
(38.09169492630212,
 array([-1.19443447e-01,  4.47799511e-02,  5.48526168e-03,  2.34080361e+00,
        -1.61236043e+01,  3.70870901e+00, -3.12108178e-03, -1.38639737e+00,
         2.44178327e-01, -1.09896366e-02, -1.04592119e+00,  8.11010693e-03,
        -4.92792725e-01]))
```

```
y_train_pred = reg.predict(X_train)
rmse = np.sqrt(MSE(y_train, y_train_pred))
print('RMSE (train):', rmse)
```

```
RMSE (train): 4.396188144698282
```

```
y_test_pred = reg.predict(X_test)
rmse = np.sqrt(MSE(y_test, y_test_pred))
print('RMSE (test):', rmse)
```

```
RMSE (test): 5.78350931508513
```

The LinearRegression Class

Northeastern University

- The score() method of the regressor returns the R^2 score:

```
reg.score(X_train, y_train)
```

```
0.7730135569264234
```

```
reg.score(X_test, y_test)
```

```
0.5892223849182509
```

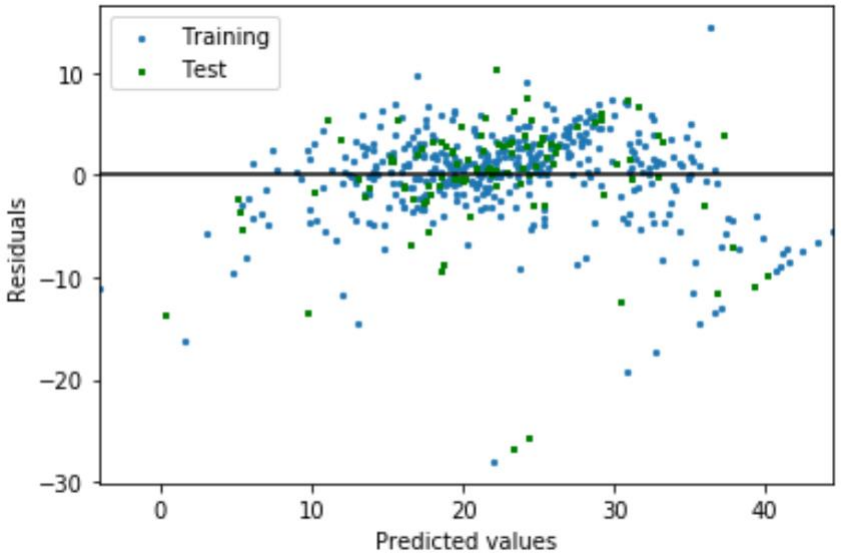
Analyzing Regression Errors

Northeastern University

- In addition to computing the performance measures, we can also plot the residuals vs. the predicted values

```
def plot_residuals(y_train_pred, y_train, y_test_pred, y_test):
    plt.scatter(y_train_pred, y_train_pred - y_train,
                s=5, marker='o', label='Training')
    plt.scatter(y_test_pred, y_test_pred - y_test,
                s=5, marker='s', c='g', label='Test')
    plt.xlabel('Predicted values')
    plt.ylabel('Residuals')
    plt.legend()
    xmin = min(y_train_pred.min(), y_test_pred.min())
    xmax = max(y_train_pred.max(), y_test_pred.max())
    plt.hlines(y=0, xmin=xmin, xmax=xmax)
    plt.xlim(xmin, xmax)
```

```
plot_residuals(y_train_pred, y_train, y_test_pred, y_test)
```



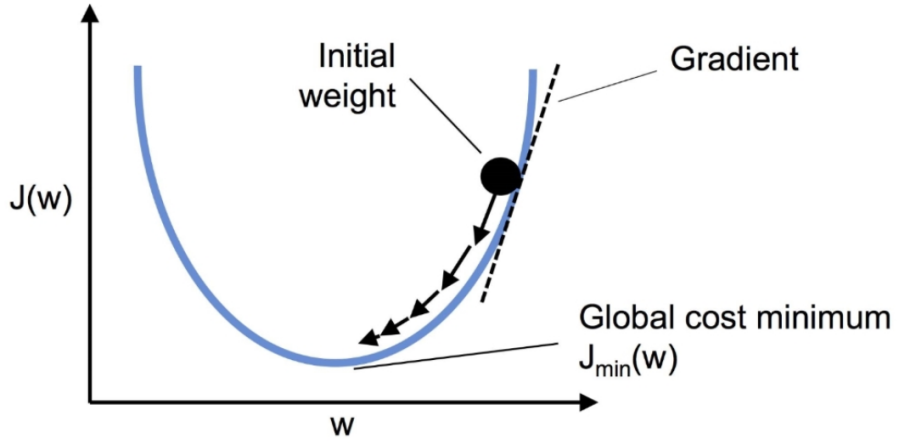
Issues with the Closed Form Solution

Northeastern University

- Inefficient when there is a large number of features
  - Requires the computation of the inverse of X^TX, which is an $d \times d$ matrix
    - Typically involves $O(d^3)$ operations
- Requires to hold the entire data matrix in memory
- Doesn't support incremental (online) learning
- Alternative approach: gradient descent!

Gradient Descent

- **Gradient descent** (GD) is an iterative approach for minimizing a function
- At each step, GD measures the gradient of the loss function with respect to **w**
  - The gradient points in the direction of steepest increase in the loss function
- Then, GD takes a step in the direction of the negative gradient
- The algorithm continues until it converges to a minimum (where the gradient is zero)



Gradient Descent Algorithm

Northeastern University

- The general algorithm for finding the minimum of a loss function using GD:
  - Start with some initial **w**
  - Loop until convergence
    - For each w_j in **w** do:

$$w_j \leftarrow w_j - \alpha \frac{\partial}{\partial w_j} J(\mathbf{w})$$

- α is called the **learning rate** ($0 < \alpha < 1$)

Gradient Descent for Linear Regression

Northeastern University

- What is the gradient of the least squares cost function?
- Let's first work it out for the case that we have only one training example (**x**, *y*)

$$\begin{aligned}\frac{\partial}{\partial w_j} J(\mathbf{w}) &= \frac{\partial}{\partial w_j}(h_{\mathbf{w}}(\mathbf{x}) - y)^2 \\ &= 2(h_{\mathbf{w}}(\mathbf{x}) - y) \cdot \frac{\partial}{\partial w_j}(h_{\mathbf{w}}(\mathbf{x}) - y)) \\ &= 2(h_{\mathbf{w}}(\mathbf{x}) - y) \cdot \frac{\partial}{\partial w_j}\left(\sum_{k=0}^{d} w_k x_k - y\right) \\ &= 2(h_{\mathbf{w}}(\mathbf{x}) - y) x_j\end{aligned}$$

- This gives us the **LMS** (least mean squares) update rule:

$$w_j \leftarrow w_j + \alpha(y_i - h_{\mathbf{w}}(\mathbf{x}_i))x_{ij}$$

- We can group the updates of the components of **w** into an update of the vector **w**:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(\mathbf{y}_i - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i$$

Batch Gradient Descent

Northeastern University

- There are two ways to handle a training set of more than one example
- The first uses all the training data in each gradient descent step
- In this case we get the following update rule:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \sum_{i=1}^{n} (y_i - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i$$

  - The quantity in the summation is just the derivative of $J(\mathbf{w})$ for the original definition of J
- This method is called **batch gradient descent**
- In general, gradient descent may get stuck at a local minimum
- However, in linear regression the cost function $J(\mathbf{w})$ has only one global minimum, thus gradient descent always converges to the global minimum
  - assuming the learning rate α is not too large

Stochastic Gradient Descent (SGD)

Northeastern University

- There is an alternative way to batch gradient descent that also works very well
- In stochastic gradient descent we repeatedly run through the training set, and make a gradient descent step after we encounter each training example

- Loop until convergence
  - For *i* = 1 to *n*:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_i - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i$$

Stochastic Gradient Descent

Northeastern University

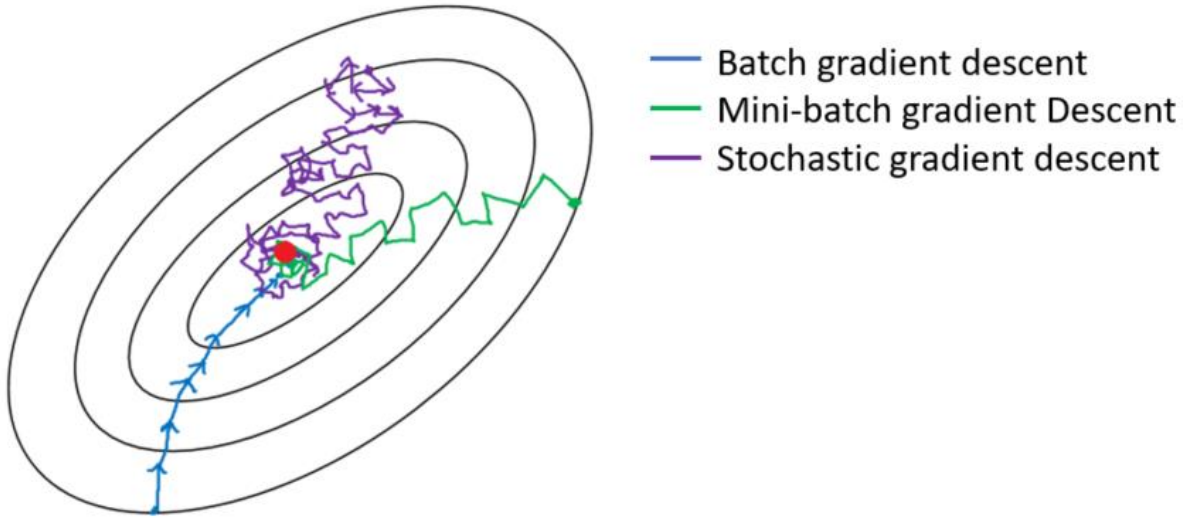
- **Advantages**:
  - Usually faster than batch gradient descent, as it makes progress after each example
  - Supports online learning, where new data are coming in one at a time
  - Introduces randomness that allows to escape from local minima
- **Disadvantages**:
  - Convergence to the minimum is not guaranteed
    - But in practice gets very close to the true minimum

- SGD is often preferred over batch gradient descent, especially for large training sets

Mini-Batch Gradient Descent

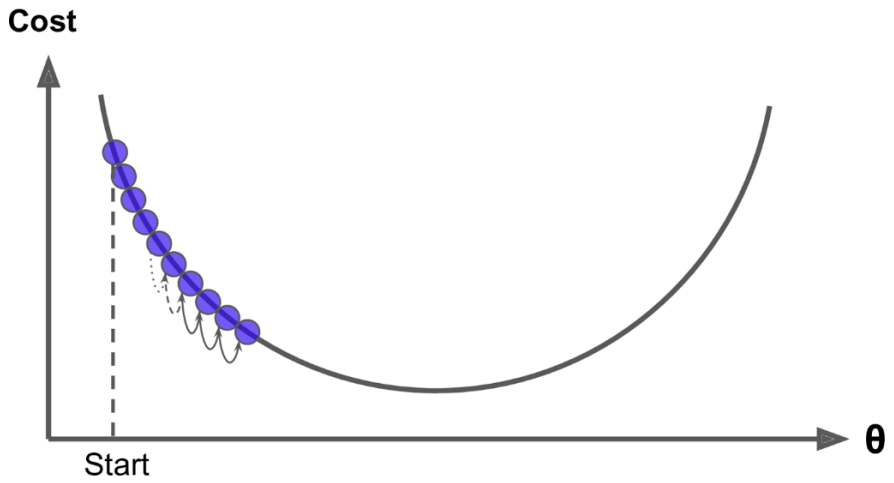
Northeastern University

- A compromise between batch gradient descent and SGD
- Gradient updates are performed on subsets of the training set called **mini-batches**
  - Each mini-batch is typically between 10 and 1,000 examples, chosen at random
- Reduces the noise in SGD, but is still more efficient than full-batch updates

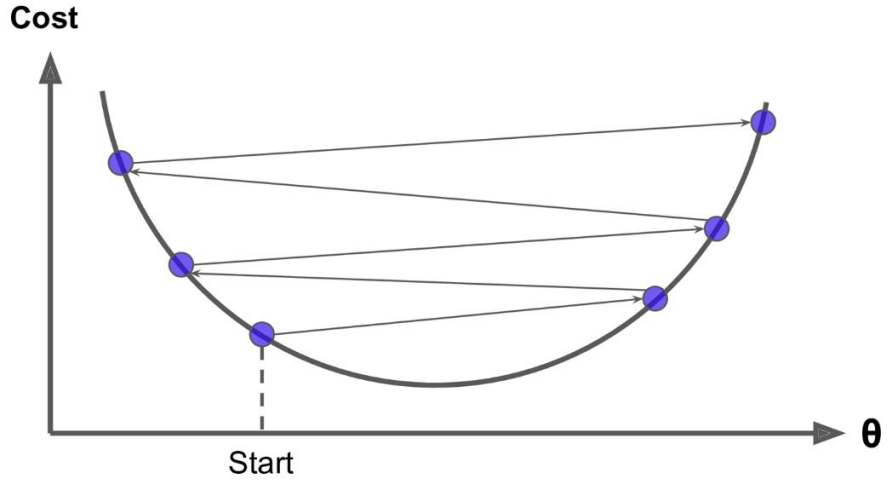


Learning Rate

- The learning rate should be carefully tuned
- If you pick a learning rate that is too small, learning will take too long
- Conversely, if you specify a learning rate that is too large, the weight vector may bounce perpetually across the bottom of the well



Learning rate too small



Learning rate too high

Learning Rate Schedule

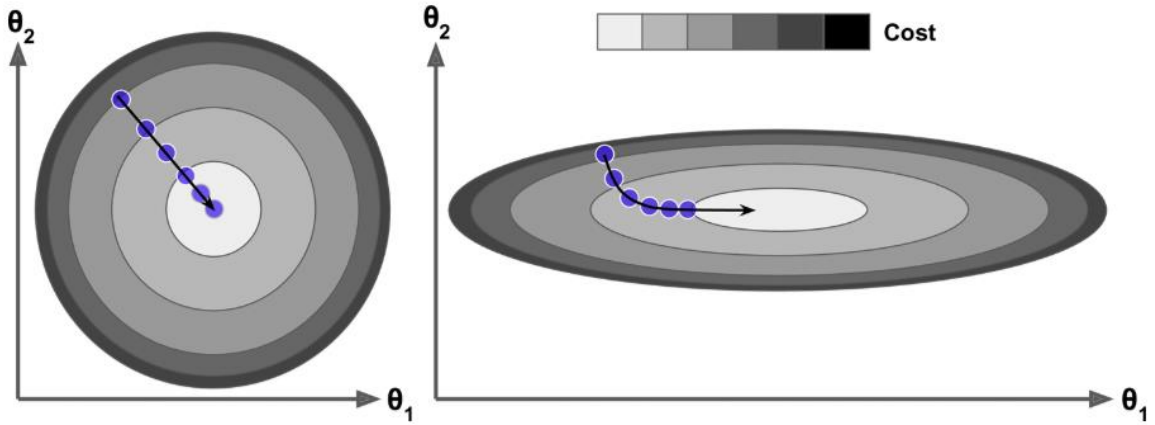
Northeastern University

- By slowly decreasing the learning rate, we can ensure that the parameters will converge to the global minimum rather than oscillate around it
  - The steps start out large, which helps make quick progress and escape local minima
  - Then the steps get smaller and smaller, allowing GD to settle at the global minimum
- **Learning rate schedule** is a function that regulates the learning rate at each iteration
- Common learning rate schedule functions:
  - **Time-based decay** $\alpha = \frac{\alpha_0}{1 + k \cdot t}$
    - α_0 and k are hyperparameters, t is the iteration number
  - **Exponential decay** $\alpha = \alpha_0 \cdot e^{-kt}$
  - **Adaptive** – keep the learning rate constant as long as the training error keeps decreasing. If for k iterations the error doesn't drop, the current learning rate is divided by some factor (e.g., 5).

Feature Scaling

Northeastern University

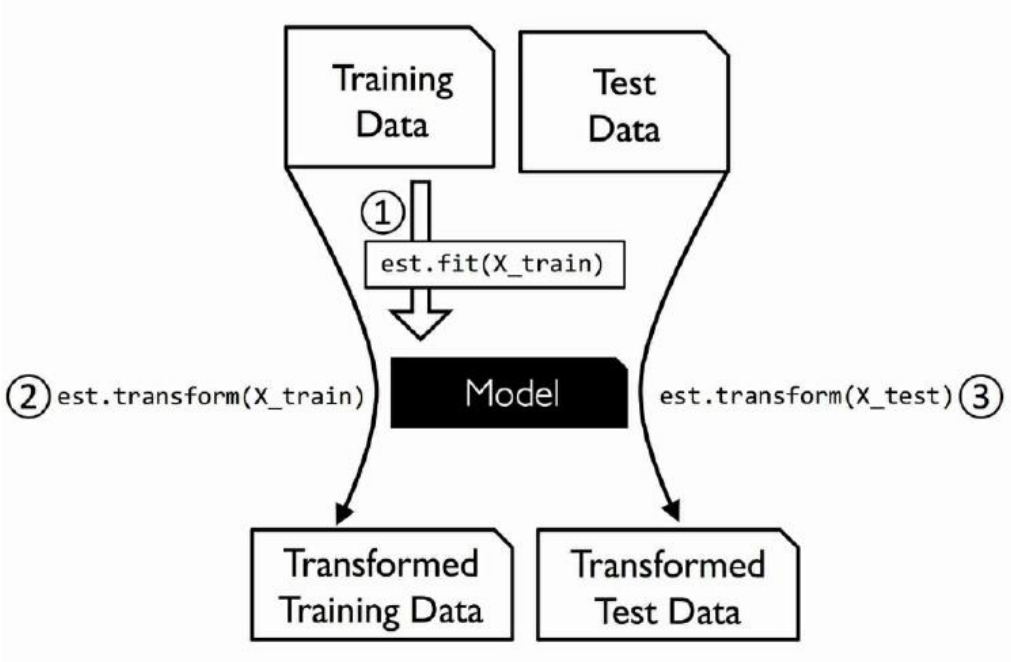
- When using gradient descent, you should scale all your features
- Otherwise it might not converge, or converge very slowly
  - Since the learning rate may be too high in some dimensions and too slow in others



Scikit-Learn Transformers

Northeastern University

- Scikit-learn provides a library of transformers, which may clean, reduce, expand or generate feature representations
  - For a full list of transformers check https://scikit-learn.org/stable/data_transforms.html
- Transformers provide the following methods:
  - **fit()** - learns model parameters from the data set
    - e.g., mean and std for normalization
  - **transform()** - applies the transformation to the data
  - **fit_transform()** - calls both fit() and transform()



Feature Scaling using a Transformer

Northeastern University

- **StandardScaler** is a transformer that standardizes features by removing the mean and scaling to unit variance

$$x' = \frac{x - \mu}{\sigma}$$

- Example:

```
from sklearn.preprocessing import StandardScaler

data = [[1, 1, 1, 1, 1],
        [2, 5, 10, 50, 100],
        [3, 10, 20, 150, 200]]

scaler = StandardScaler()
scaler.fit_transform(data)
```

```
array([[-1.22474487, -1.1769647 , -1.20270298, -1.06442383, -1.22268823],
       [ 0.        , -0.09053575, -0.04295368, -0.27416977, -0.00410298],
       [ 1.22474487,  1.26750044,  1.24565666,  1.33859361,  1.22679121]])
```

SGDRegressor

Northeastern University

- SciKit-Learn's **SGDRegressor** implements linear regression using SGD

class `sklearn.linear_model.` **`SGDRegressor`**(*loss='squared_loss', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, random_state=None, learning_rate='invscaling', eta0=0.01, power_t=0.25, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, warm_start=False, average=False*) [source]

Parameter	Description
loss	The loss function to be used. The possible values are 'squared_loss' (default), 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'.
penalty	The penalty (aka regularization term) to be used.
max_iter	The maximum number of passes over the training data.
tol	The stopping criterion. If it is not None, the iterations will stop when (loss > previous_loss - tol).
shuffle	Whether or not the training data should be shuffled after each epoch. Defaults to True.
learning_rate	The learning rate schedule: 'constant': eta = eta0 'optimal': eta = 1.0 / (alpha * (t + t0)) [default] 'invscaling': eta = eta0 / pow(t, power_t)

SGDRegressor Example

Northeastern University

- The following code runs a linear regression using SGD on the Boston housing data
  - The data must be scaled first

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
from sklearn.linear_model import SGDRegressor

reg2 = SGDRegressor()
reg2.fit(X_train_scaled, y_train)

reg2.intercept_, reg2.coef_
```

```
(array([22.62585137]),
 array([-0.90530238,  0.95403481, -0.15796404,  0.63477362, -1.70517082,
         2.60622292, -0.11376033, -2.84979078,  1.60423115, -1.29452695,
        -2.24204858,  0.69706245, -3.53684018]))
```

SGDRegressor Example

Northeastern University

- The RMSE on the training and test sets is:

```
y_train_pred = reg2.predict(X_train_scaled)
rmse = np.sqrt(MSE(y_train, y_train_pred))
print('RMSE (train):', rmse)
```

```
RMSE (train): 4.402188999208474
```

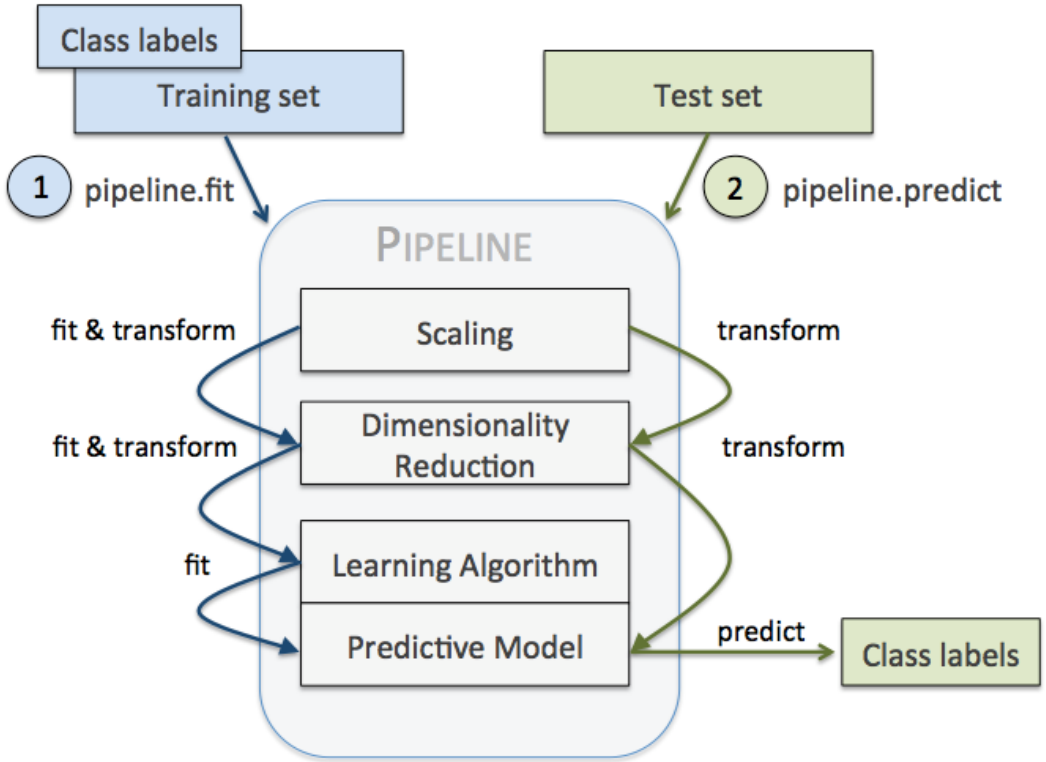
```
X_test_scaled = scaler.transform(X_test)
y_test_pred = reg2.predict(X_test_scaled)
rmse = np.sqrt(MSE(y_test, y_test_pred))
print('RMSE (test):', rmse)
```

```
RMSE (test): 5.786732494580718
```

Scikit-Learn Pipelines

Northeastern University

- Pipelines allow you to chain multiple transformers with a final estimator
- The pipeline can be used as any other estimator



Scikit-Learn Pipelines

Northeastern University

- For example, we can define a pipeline that combines the StandardScaler with the regression estimator:

```
from sklearn.pipeline import Pipeline

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('reg', SGDRegressor())
])

pipe.fit(X_train, y_train)
```

```
Pipeline(steps=[('scaler', StandardScaler()), ('reg', SGDRegressor())])
```

```
y_test_pred = pipe.predict(X_test)

rmse = np.sqrt(MSE(y_test, y_test_pred))
print('RMSE (test):', rmse)
```

```
RMSE (test): 5.861691812366359
```

Scikit-Learn Pipelines

Northeastern University

- You can access the components of a pipeline using its **steps** attribute:

```
pipe.steps[1]
```

```
('reg', SGDRegressor())
```

- Or via its **named_steps**:

```
pipe.named_steps['reg'].intercept_
```

```
array([22.6115222])
```

- Parameters of the estimators are accessed via **<estimator>__<parameter>**:

```
pipe.set_params(reg__max_iter=2000)
```

```
Pipeline(steps=[('scaler', StandardScaler()),
                ('reg', SGDRegressor(max_iter=2000))])
```

Probabilistic Interpretation

Northeastern University

- Given a regression problem, why linear regression, and specifically the least squares cost function *J*, are reasonable choices?
- We can show that under a set of probabilistic assumptions, least-squares regression is derived as a very natural algorithm
- We first introduce an important principle in statistics and machine learning called **maximum likelihood**

Maximum Likelihood Estimation (MLE)

Northeastern University

- A method for finding the parameters of a probabilistic model that maximize the likelihood of the observations
- Assume that we have a set of *n* data points $X = \{x_1, \ldots, x_n\}$ drawn from some probability distribution $P(X; \theta)$ with unknown parameters θ
- The **likelihood** of θ is defined as the probability of obtaining the data given θ

$$\mathcal{L}(\theta|X) = p(X|\theta)$$

- Assuming the points are identically and independently distributed (i.i.d.), we write:

$$\mathcal{L}(\theta|X) = p(x_1, \ldots, x_n|\theta) = \prod_{i=1}^{n} p(x_i|\theta)$$

- For practical reasons, the **log likelihood** is more commonly used:

$$\ell(\theta|X) = \log \mathcal{L}(\theta|X) = \sum_{i=1}^{n} \log(p(x_i|\theta))$$

- Our goal is to find the parameters θ that **maximize** the likelihood function

Maximum Likelihood Example

Northeastern University

- Assume that we have *n* points generated from 1D Gaussian distribution and we would like to find the parameters of this distribution (μ, σ)
- The likelihood of the parameters μ and σ given the data is:

$$\mathcal{L}(\mu, \sigma|X) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x_i - \mu)^2}{2\sigma^2}\right)$$

- Thus, the log likelihood is:

$$\begin{aligned}
\ell(\mu, \sigma|X) &= \sum_{i=1}^{n} \log\left[\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x_i - \mu)^2}{2\sigma^2}\right)\right] \\
&= n \log \frac{1}{\sqrt{2\pi}\sigma} - \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} \\
&= -\sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} - n \log \sigma - \frac{n}{2} \log 2\pi
\end{aligned}$$

Maximum Likelihood Example

Northeastern University

- To find the parameters μ and σ that yield the maximum likelihood we can take the derivatives of the log likelihood with respect to them and set them to 0:

$$\frac{\partial \ell}{\partial \mu} = -\sum_{i=1}^{n} \frac{-2(x_i - \mu)}{2\sigma^2} = \frac{1}{\sigma^2} \sum_{i=1}^{n} (x_i - \mu) = 0$$

$$\Rightarrow \sum_{i=1}^{n} x_i - n\mu = 0$$

$$\Rightarrow \mu = \frac{\sum_{i=1}^{n} x_i}{n}$$

$$\frac{\partial \ell}{\partial \sigma} = -\sum_{i=1}^{n} \frac{-2(x_i - \mu)^2}{2\sigma^3} - \frac{n}{\sigma} = \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{\sigma^3} - \frac{n}{\sigma} = 0$$

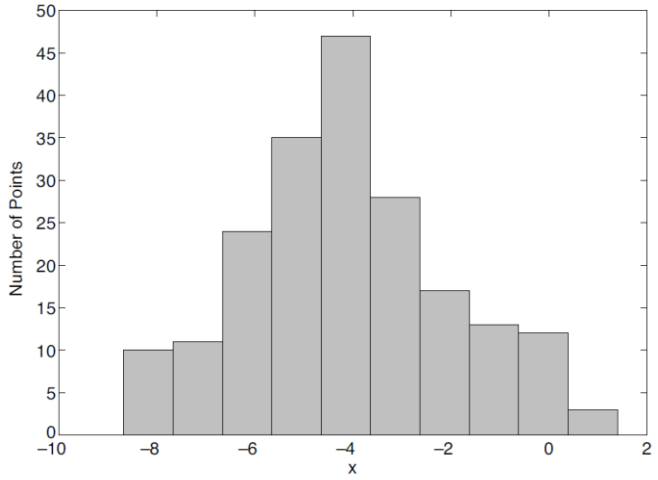
$$\Rightarrow \frac{1}{\sigma^3} \sum_{i=1}^{n} (x_i - \mu)^2 = \frac{n}{\sigma}$$

$$\Rightarrow \sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \mu)^2}{n}}$$

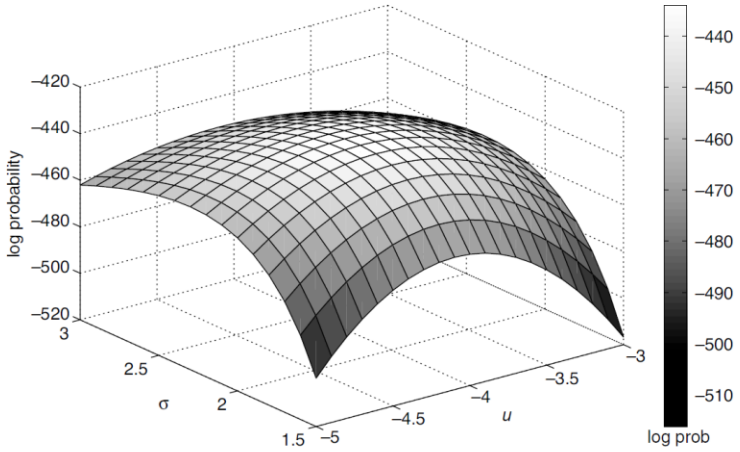
- **Conclusion**: the maximum likelihood mean is the mean of the given points, and the maximum likelihood standard deviation is the standard deviation of the points

Maximum Likelihood Example

- Suppose that we have a set of 200 points whose histogram is shown on the left
- The parameters that maximize the log likelihood are the mean and standard deviation of the 200 points: μ = -4.1, σ = 2.1
- These are close to the parameters of the underlying Gaussian: μ = -4.0, σ = 2.0



(a) Histogram of 200 points from a Gaussian distribution.



(b) Log likelihood plot of the 200 points for different values of the mean and standard deviation.

Class Exercise

Northeastern University

- There are 5 balls in a bag. Each ball is either red or blue.
- Let θ be the number of blue balls
- We want to estimate θ, so we draw 4 balls **with replacement** out of the bag, replacing each one before drawing the next
- We get "blue", "red", "blue", and "blue" (in that order)

- What is the likelihood of getting exactly that sequence of colors (expressed as a function of θ)?
- What is the maximum likelihood estimate for θ? Show your computation

Maximum Likelihood and Least Squares

- The least squares cost function can be motivated as a maximum likelihood solution under an assumed Gaussian noise model
- We assume that the target variables and the inputs are related via the equation:

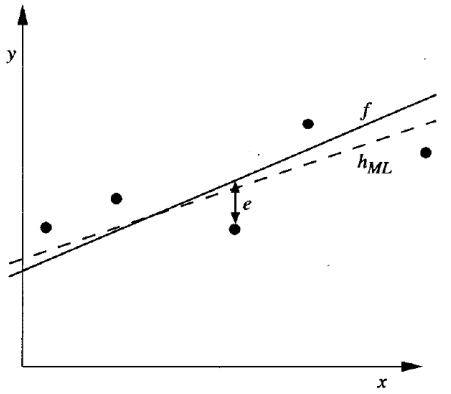
$$y_i = \mathbf{w}^T \mathbf{x}_i + \epsilon_i$$

  - ε_i is an error term that captures either unmodeled effects or some random noise
- We further assume that the errors are distributed i.i.d. according to a normal distribution with zero mean and some variance, i.e., $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$
- The density of ε_i is given by

$$p(\epsilon_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\epsilon_i^2}{2\sigma^2}\right)$$

- Therefore

$$p(y_i|\mathbf{x}_i, \mathbf{w}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - \mathbf{w}^T \mathbf{x}_i)^2}{2\sigma^2}\right)$$



Maximum Likelihood and Least Squares

Northeastern University

- Therefore, the likelihood of **w** is: $\mathcal{L}(\mathbf{w}) = \mathcal{L}(\mathbf{w}|X, \mathbf{y}) = p(\mathbf{y}|X, \mathbf{w})$
- By the independence assumption on the errors (and hence also the *y*'s):

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^{n} p(y_i|\mathbf{x}_i, \mathbf{w}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - \mathbf{w}^T\mathbf{x}_i)^2}{2\sigma^2}\right)$$

- The log likelihood of **w** is:

$$\begin{aligned}
\ell(\mathbf{w}) &= \log \mathcal{L}(\mathbf{w}) \\
&= \log \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - \mathbf{w}^T\mathbf{x}_i)^2}{2\sigma^2}\right) \\
&= \sum_{i=1}^{n} \log\left(\frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(y_i - \mathbf{w}^T\mathbf{x}_i)^2}{2\sigma^2}\right]\right) \\
&= -n\log(\sqrt{2\pi}\sigma) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i)^2
\end{aligned}$$

- Hence, maximizing the likelihood of **w** is the same as minimizing $\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i)^2$

Linear Regression Summary

Northeastern University

Algorithm	Large training set	Many features	Out-of-core support	Hyper-parameters	Scaling required	Scikit-Learn
Normal equations	Fast	Slow	No	0	No	LinearRegression
Batch GD	Slow	Fast	No	2	Yes	N/A
Stochastic GD	Fast	Fast	Yes	≥2	Yes	SGDRegressor
Mini-batch GD	Fast	Fast	Yes	≥2	Yes	N/A